Promela Planning

Stefan Edelkamp

Institut für Informatik
Georges-Köhler-Allee 51
D-79110 Freiburg
edelkamp@informatik.uni-freiburg.de

Abstract. In this paper a compiler from a restricted subset of SPIN's input language Promela into an action planning description language is presented. It exploits the representation of protocols as communicating finite state machines.
The work targets the transfer between the state space exploration areas AI planning and model checking by making protocols accessible for action planners.
It provides a practical study of the expressiveness of different input specifications, introduces concurrency and quality metrics to counterexample traces, and compares search, pruning and acceleration methods.
Indirectly, refined estimates for improved error detection in directed protocol validation are introduced. For example, the *relaxed plan heuristic*, which comes along with an *enforced hill climbing* search engine.
The experimental results are encouraging. In some sample protocols, planners perform close to state-of-the-art model checkers.

1 Introduction

Communication protocols [17] are concurrent software systems with main purpose to organize information exchange between individual processes. Due to interleavings of process execution traces and the communication load itself, the number of global system states is large even for simple and moderate sized protocol specifications. By this combinatorial growth, called the *state explosion* problem [4], many protocol designs contain subtle concurrency bugs.

Therefore, in the design process automated procedures are needed to certify that stated assertions or global invariants are valid, and that no deadlock occurs. Validating these kinds of properties corresponds to solving a reachability problem in the state space graph with respect to a set of designated error states.

Probably the most effective technique to establish errors fast is heuristic search [22]: a guided traversal of the state space, that exploits information of the error description to focus the search. There are many possibilities to devise an evaluation function and a corresponding search routine. A*-like [13], and IDA*-like [19] engines incorporate the estimated path length into the evaluation function $f(n) = \lambda g(n) + (1 - \lambda)h(n)$, where $g(n)$ is the generating path length to state n and h is the approximated distance from n to the goal. For $\lambda \geq 0.5$ and lower bound h, obtained solutions are provably optimal.

Recent advances in applying genetic algorithms (GAs) to model checking [12] exploit evaluation functions on execution paths. The population is a set of paths in the

state space graph, which through the basic transformations recombination, selection and mutation are tuned towards an error trail.

Setting the evaluation to the estimated distance of the end of the path to the goal links GAs to heuristic search exploration, with the difference that the former improves a set of samples in the solution space, while the latter systematically enumerates it.

The evaluation functions that have been applied so far exploit the Hamming distance of the state-vector [23], the number of active processes, the accumulated shortest path distances of each process state, and the tree expansion of the error formulae [7]. The latter heuristic estimate $H_f(S)$ predicts the number of transitions necessary to establish the truth of failure function f and is recursively defined on the structure of f. The estimate assumes independence of the events and neglects correlations between them.

To make more informed heuristic estimates accessible for the exploration of communication protocols, in this paper we contribute a compiler from Promela into recent problem domain description languages developed for action planning, capable to deal with numerical quantities and plan metrics, called PDDL2.1 [9]. Since we consider safety problems only, a successful plan is a trail to one of a set of error states. Planners usually do not prove unsolvability.

The parser does not yet feature full language expressiveness of Promela, e.g. it requires processes to be active, but for many problem specifications at hand, it appears to be a sufficiently large basis. It avoids internal access to the Spin validator, which can turn out to be a burden for the unexperienced programmer. Both process communication forms (queues and shared variables) are supported. The approach utilizes the Spin parser as a welcome pre-processor subcomponent. Similar to the XSpin interface, it starts with the process automata file.

Since only active processes were allowed, the state description in terms of queue width and number of created processes is fixed. Compared to the other proposed estimates, this assumption is not restrictive, but when tackling general software verification problems [18], the approach definitely has to be extended.

The research is motivated by prior work in planning as heuristic search [3]. The estimate we discuss is integral part of Metric-FF [15], MIPS [6], and LPG [11], the best fully automatic planners in the last international planning system competition.

The paper contributes feasibility and practicability of using a planner to model check protocols. In other word, it produces very first data for a *model checking via action planning* paradigm. With uncertainty on the initial state and transition reliability, this includes the application of conformant planners to perform conformance checking.

The paper is structured as follows. First we give a fixed-length state space characterization of a communication protocol system to exploit the structure of the underlying exploration problem. Then the action planning formalism STRIPS and PDDL2.1 and their state space characterizations are introduced. Afterwards we examine the stages of the compiling process in detail, taking the Leader Election protocol as the running example. Next we introduce the plan relaxation heuristic, which solves a polynomial relaxation of the exploration problem on-the-fly for each encountered state. We close with drawn conclusions and a discussion on appropriate extensions of the approach.

2 State Space Representation

To model a communication system, we choose the following representation[1].

We assume each communication process P to be represented as a labeled finite state graph $G(P) = (S(P), \Sigma(P), \mathit{init}(P), \mathit{curr}(P), \delta(P))$, with $S(P)$ being P's set of (local) states, $\Sigma(P)$ being the set of transitions of P, $\mathit{init}(P) \in S(P)$ (and $\mathit{curr}(P) \in S(P)$) being the initial (and current) state of P, and $\delta(P) : S(P) \times \Sigma(P) \rightarrow S(P)$ being the transition relation on P.

Similarly, we assume each queue Q to be modeled as a finite state graph $G(Q) = (S(Q), \mathit{head}(Q), \mathit{tail}(Q), \delta(Q), \mathit{mess}(Q), \mathit{cont}(Q))$, with $S(Q)$ being the set of queue states, $\mathit{head}(Q)$, $\mathit{tail}(Q) \in S(Q)$ being the head and tail states of Q, $\mathit{mess}(Q) \in M^{|S(Q)|}$ being the vector of messages in Q (M is the set of all messages), $\mathit{cont}(Q) \in \mathbb{R}^{|S(Q)|}$ being the vector of variable values in Q and $\delta(Q) : S(Q) \rightarrow S(Q)$ being the successor relation for Q; if $S(Q) = s[1], \ldots, s[k]$ then $\delta(s[i]) = s[(i+1) \bmod k]$. Explicitly modeling head and tail positions in the queue trades space for time, since queue updates reduce to constant time operations.

Shared and local variables are modeled by real numbers. The only difference of local variables compared to shared ones is the restricted visibility scope.

A state S in the global state space $\mathcal{S}$ is fully described by its process components $PC(S)$, its queue contents $QC(S)$, and the current assignments $ASS(S)$ to shared and local variables, where $PC(S)$ contains the value $\mathit{curr}(P)$ for each process P, and $QC(S)$ contains the values $\mathit{head}(Q)$, $\mathit{tail}(Q)$, and their contents $\mathit{mess}(Q)$, $\mathit{cont}(Q)$ for each queue Q. Given a fixed number of processes np and queues nq, a constant number of local variables $nl(P)$ for each process P, and a constant number of shared variables ns, the global system state space $\mathcal{S}$ can be expressed as

$$\begin{aligned}\mathcal{S} = {} & S(P_1) \times \ldots \times S(P_{np}) \times S(Q_1) \times \ldots \times S(Q_{nq}) \times S(Q_1) \times \ldots \times S(Q_{nq}) \times \\ & M^{|S(Q_1)|} \times \ldots \times M^{|S(Q_{nq})|} \times \mathbb{R}^{|S(Q_1)|} \times \ldots \times \mathbb{R}^{|S(Q_{nq})|} \times \\ & \mathbb{R}^{nl(P_1)} \times \ldots \times \mathbb{R}^{nl(P_{np})} \times \mathbb{R}^{ns},\end{aligned}$$

or as $\mathcal{S} \subset \mathbb{R}^l$, for some fixed value $l \in \mathbb{N}$, when identifying process/queue states and messages with their index. Note that in this characterization the state space is not necessarily finite. Having fixed the set of states, we still need to explain the set of transitions we allow. Transitions are specified with respect to the set of processes they are declared in. Therefore, we might consider the following fundamental set of operations (in Promela-like notation):

`queue!message(variable)` The transition writes compound (`message` and `variable`) into the `queue` at the position of the tail pointer. The returned value of the operation is true if successful, false otherwise.

`queue?message(variable)` If `message` matches the one at the head in `queue` the transition reads the according contents to `variable`. The returned value is true if successful, false otherwise.

[1] For the sake of brevity in the presentation we neglect advanced modeling aspects such as rendez-vous communication and (in-)validity of end-states.

`variable = expression` The evaluated `expression` is assigned to the according (local/shared) variable, where `expression` is a formula tree on different variables and constants.

`if (variable == condition) body1 else body2` Continue the execution with `body1`, if `variable` equals `condition`, and with `body2`, otherwise.

Since sequential composition, selection and assignments are already sufficient to model abacus and Turing machine computation, the state space problem for protocols that we have devised so far is in fact undecidable. Explicit model checkers like Spin bypass the problem by restricting the range of the variables to a finite domain, resulting into a large but finite state space graph.

According to the finite state representation of a process, all transition can be specified in form of preconditions and effects. For transition $Q!m(v)$ from state s_1 to s_2 of process P, we have preconditions $\mathit{curr}(P) = s_1$ and effects $\mathit{tail}(Q) \leftarrow (\mathit{tail}(Q) + 1) \bmod |S(Q)|$, $\mathit{mess}(Q, \mathit{tail}(Q)) \leftarrow m$, $\mathit{cont}(Q, \mathit{head}(Q)) \leftarrow v$, and $\mathit{curr}(P) = s_2$. For the transition $Q?m(v)$ from state s_1 to s_2 of process P we have: If $\mathit{mess}(Q)[\mathit{head}(Q)] = m$, and $\mathit{curr}(P) = s_1$ then $\mathit{curr}(P) = s_2$, $v \leftarrow \mathit{cont}(Q)[\mathit{head}(Q)]$, and $\mathit{head}(Q) \leftarrow \delta(Q)(\mathit{head}) = (\mathit{head}(Q) + 1) \bmod |S(Q)|$. Transition $v = exp$ from state s_1 to s_2 of process P reads as: given $\mathit{curr}(P) = s_1$, set $\mathit{curr}(P) = s_2$ and $v \leftarrow exp$. Transition like $\mathit{if}(v = exp)$ link state s_1 to s_2 in process P, if $\mathit{curr}(P) = s_1$ and v equals exp, so that $\mathit{curr}(P) = s_2$ is set.

3 Propositional and Numerical Planning

We see that transitions in protocol verification are in fact operators with global *preconditions* and *effects*, much in the sense that if the conditions of a transitions are satisfied in the current state, it is executed by performing its effects. This leads to a small excursion in action planning terminology. A grounded STRIPS [8] task considers a set of actions of type $a = (\mathit{pre}(a), \mathit{eff}(a)^+, \mathit{eff}(a)^-)$. The result of applying a in state S is a state $S \cup \mathit{eff}(a)^+ \setminus \mathit{eff}(a)^-$ if $\mathit{pre}(a) \subseteq S$. The state space $\mathcal{S}$ is the power set of grounded predicates, so that each $S \in \mathcal{S}$ can be characterized as a vector in $\mathbb{B}^l$, for a fixed value $l \in \mathbb{N}$.

One of the issues that are currently in the focus of AI planning research are numerical conditions and effects. In the extended formalism PDDL2.1 [9], numerical conditions are of the form $e \oplus e'$ with $\oplus \in \{=, \leq, <, \geq, >\}$, where e, e' are arithmetic and boolean expressions trees over the set of variables, and assignments are of the form $v \otimes e$ with $\otimes \in \{\texttt{:=}, \texttt{+=}, \texttt{-=}\}$, where v is a variable. The representation of the search space as

$$\begin{aligned}\mathcal{S} = {} & \mathbb{B}^{|S(P_1)|} \times \ldots \times \mathbb{B}^{|S(P_{np})|} \times \\ & \mathbb{B}^{|S(Q_1)|} \times \ldots \times \mathbb{B}^{|S(Q_{nq})|} \times \mathbb{B}^{|S(Q_1)|} \times \ldots \times \mathbb{B}^{|S(Q_{nq})|} \times \\ & \mathbb{R}^{|S(Q_1)|} \times \ldots \times \mathbb{R}^{|S(Q_{nq})|} \times \mathbb{R}^{|S(Q_1)|} \times \ldots \times \mathbb{R}^{|S(Q_{nq})|} \times \\ & \mathbb{R}^{nl(P_1)} \times \ldots \times \mathbb{R}^{nl(P_{np})} \times \mathbb{R}^{ns}.\end{aligned}$$

```
state  1 -(tr  3)-> state  2 line 38 => Active[_pid] = 1
state  2 -(tr  4)-> state  3 line 39 => know_winner[_pid] = 0
state  3 -(tr  5)-> state  4 line 40 => maximum[_pid] = 0
state  4 -(tr  6)-> state  5 line 41 => mynumber[_pid] = 0
state  5 -(tr  7)-> state 51 line 43 => q[((_pid+1)%2)]!one,mynumber[_pid]
state 51 -(tr  8)-> state 18 line 44 => q[_pid]?one,nr[_pid]
state 51 -(tr 16)-> state 36 line 44 => q[_pid]?two,nr[_pid]
state 51 -(tr 22)-> state 43 line 44 => q[_pid]?winner,nr[_pid]
state 18 -(tr  9)-> state 14 line 46 => ((Active[_pid]>0))
state 18 -(tr  2)-> state 17 line 46 => else
state 14 -(tr 10)-> state  9 line 48 => ((nr[_pid]!=maximum[_pid]))
state 14 -(tr  2)-> state 12 line 48 => else
state  9 -(tr 11)-> state 10 line 50 => q[((_pid+1)%2)]!two,nr[_pid]
state 10 -(tr 12)-> state 51 line 51 => neighbourR[_pid] = nr[_pid]
state 12 -(tr 13)-> state 13 line 53 => know_winner[_pid] = 1
state 13 -(tr 14)-> state 51 line 54 => q[((_pid+1)%2)]!winner,nr[_pid]
state 17 -(tr 15)-> state 51 line 57 => q[((_pid+1)%2)]!one,nr[_pid]
state 36 -(tr  9)-> state 32 line 61 => ((Active[_pid]>0))
state 36 -(tr  2)-> state 35 line 61 => else
state 32 -(tr 17)-> state 28 line 63 => ((neighbourR[_pid]>nr[_pid]))
state 32 -(tr  2)-> state 31 line 63 => else
state 28 -(tr 18)-> state 24 line 65 => ((neighbourR[_pid]>maximum[_pid]))
state 28 -(tr  2)-> state 27 line 65 => else
state 24 -(tr 19)-> state 25 line 67 => maximum[_pid] = neighbourR[_pid]
state 25 -(tr 20)-> state 51 line 68 => q[((_pid+1)%2)]!one,neighbourR[_pid]
state 27 -(tr 21)-> state 51 line 70 => Active[_pid] = 0
state 31 -(tr 21)-> state 51 line 73 => Active[_pid] = 0
state 35 -(tr 11)-> state 51 line 76 => q[((_pid+1)%2)]!two,nr[_pid]
state 43 -(tr 23)-> state 48 line 79 => ((nr[_pid]!=mynumber[_pid]))
state 43 -(tr  2)-> state 41 line 79 => else
state 48 -(tr 26)-> state 54 line 85 => ((know_winner[_pid]>0))
state 48 -(tr  2)-> state 47 line 85 => else
state 54 -(tr 27)-> state  0 line 91 => -end-
state 47 -(tr 14)-> state 54 line 87 => q[((_pid+1)%2)]!winner,nr[_pid]
state 41 -(tr 24)-> state 42 line 82 => nr_leaders[0] = (nr_leaders[0]+1)
state 42 -(tr 25)-> state 48 line 83 => assert((nr_leaders[0]==1))
```

Fig. 1. The automata description file for the Leader Election protocol.

indicates that protocol validation problems can be casted as planning problems with propositional and numerical state information, where the propositions and numbers are indexed by process identifiers, by local and shared variables as well as by queue identifiers. The propositions we use are *at*(P, s) for $S(P) = s$, *head*(Q, s) for *head*$(Q) = s$, and *tail*(Q, s) for *tail*$(Q) = s$. For the numerical propositions we choose *mess*(Q, s) for *mess*$(Q)[s]$, and *cont*(Q, s) for *cont*$(Q)[s]$ Unfortunately, accesses like *mess*$(Q)[(head(Q)+1)$ *mod* $|S(Q)|]$ are no longer available. Therefore, we take the current queue position q_1 and q_2 as additional parameters, where we know that $q_2 = \delta(Q, q_1)$. E.g, for $Q?m(v)$ this results to the formula: If *head*(Q, q_1), *mess*$(Q, q_1) = m$, and *curr*(P, s_1) then $\neg$*curr*(P, s_1), $\neg$*head*(P, q_1), *curr*(P, s_2), *head*(P, q_2), and *value*$(v) \leftarrow$ *cont*(Q, q_1).

The representional blow-up due to the binary encoding of process and communication queue states and the upgrade of variables and messages to $\mathbb{R}$ is not as critical as the first glance may indicate. It can be compensated by static analyzer tools featured by planners like Stan [21] and Discoplan [10] that efficiently detect and eliminate constant predicate and partition atoms into groups of mutually exclusive facts.

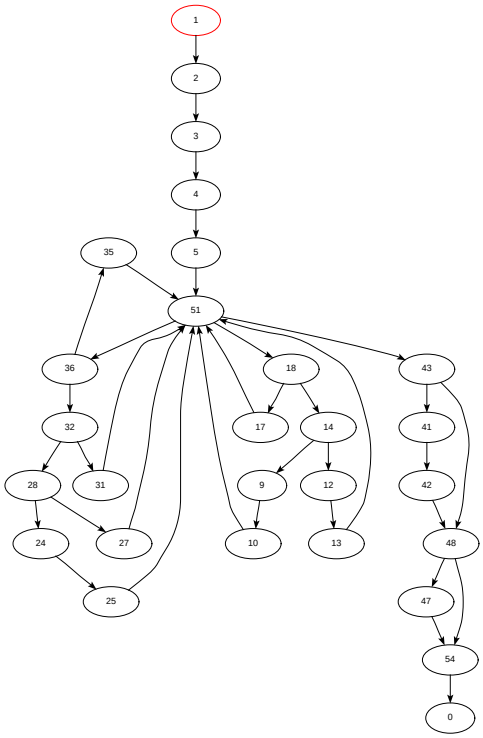

Fig. 2. The automata of Figure 1 taken from the XSpin interface.

4 Compilation Process

As main input for our parser, instead of the Promela file itself, we start with the automata representation that is produced by Spin, i.e. we take the Promela input file, generate the corresponding c-file, and run the executable with option `-d`. We avoid state merging by setting parameter `-o3` as a option to Spin. The process automata specification for the Leader Election protocol (with two `node` processes and two queues `q` of size 4) is visualized in Figure 2 with full text representation in Figure 1.

As said, we assume all processes to be active. This imposes some restrictions to the Promela model. However, in our set of benchmark problems, all Promela processes (`proctypes`) are invoked by a single init process, such that it is not difficult to provide the entire benchmark set in this format. The corresponding Promela simplifications are more or less textual substitutions, which presumably can be provided by automated procedures. So far these simplifications have been performed manually. Even for involved examples this step is not a burden even for an untrained Spin user. To deal with different

```
(:action activate-trans
   :parameters (?p - process ?t - transition ?s1 ?s2 - state)
   :precondition (and (trans ?t ?s1 ?s2) (at ?p ?s1))
   :effect (and (activate ?p ?t)))

(:action perform-trans
   :parameters (?p - process ?t - transition ?s1 ?s2 - state)
   :precondition (and (trans ?t ?s1 ?s2) (ok ?p ?t) (at ?p ?s1))
   :effect (and (at ?p ?s2) (not (at ?p ?s1)) (not (ok ?p ?t))))
```

Fig. 3. Preparing and executing a process state transition.

process parameters, we highlight that in Spin each process can be accessed by process identifier `_pid`. The value is continuously increased for each allocated process. This allows to organize the access to queues, when specifying arrays of active processes. For the ease of presentation, we have transformed all local variables into shared variable arrays, once more to be addressed by the process identifier `_pid`.

The array dimensions of process types, variables, and queues as well as queue capacity are read from the Promela input file. This is the only additional information, that is not present in the finite state representation file. To avoid conflicts with precompiler directives, we substitute all `defines` beforehand with the c-compiler command line option `-E`, which runs the precompiler only.

4.1 State Transitions

Planning problems are defined in two files, a problem domain file and an instance specific file. In the former file, predicates and actions are given, while in the latter file domain objects, their initial and goal state are specified. Only the instance file refers to grounded predicates, while actions and predicates are specified by typed object parameters. For example in the well-known *Blocks World* problem we have actions like `stack` and `unstack` with respect to two parameters of type *block* and with predicates like `on`, `on-table`, and `clear` in the preconditions and effect lists, where the instance file specifies the block configurations such as `clear(a)`, `on(a,b)`, `on(b,c)`, `on(c,t)` with respect to a set of objects, e.g. `a`, `b`, `c` of type `block` and `t` of type `table`.

For protocol modeling, we identify processes, their proctype, and a propositional description of the finite state system with states and state transitions as objects. If we have variables in the Promela specification, these are also to be found and declared as possible objects. Communication queues are defined by their channel type and content configuration. All these objects are inferred in the parser by instantiating the process identifier `_pid` with respect to the established array bounds in the automata process description file.

As an example the set of objects for the Leader Election problem with automata description of Figure 1, we have `node-0` and `node-1` as `process` objects, `node` as

```
(:action queue-read
   :parameters (?p - process ?t - transition
                ?q - queue ?v - variable)
   :precondition
      (and
        (activate ?p ?t) (settled ?q)
        (reads ?p ?q ?t) (reads-val ?p ?t ?v)
        (>= (size ?q) 1) (= (head-msg ?q) (mess ?t)))
    :effect
      (and
        (advance-head ?q) (ok ?p ?t)
        (not (activate ?p ?t)) (not (settled ?q)))
        (assign (value ?v) (head-value ?q)))
```

Fig. 4. Reading variables from a queue.

`proctype` object, `q[0]` and `q[1]` as `queue` objects, `queue-4` as `queuetype` object, `Active[pid]`, ... as `variable` objects, `state-1`, ..., `state-54` as process `state` objects and, last but not least, `Active[pid]=1` and others as state `transition` objects.

4.2 Operators

In pure message passing domains, no inference of domain file ingredients is necessary. But in the presence of local/shared variables (as in our example) the parser additionally generates action schemas for variable conditioning and change.

In the first step, we describe fixed actions applicable to all protocol domains. Note that one objective in a good model of a planning problem is to keep the number of parameters small. Grounding actions and predicates with more than five object parameters causes problems for almost any planner that we know. The reduction of parameters can best be achieved by some additional flags that are set by one action and queried by another one.

Figure 3 shows how we prepare and execute process state transitions. Action `activate-trans` activates transition t in process P if in the current state we have an option to perform t starting from s_1. Action `process-trans` triggers the transition t in process P to move from s_1 to s_2. It queries the flag `ok` which is deleted afterwards. Hence, for each transition t there is an appropriate action that performs all necessary changes according to t and that that sets the flag(s) `ok`. These operations are purely propositional.

4.3 Queue Organization

Queue organization with head and tail pointer has already been mentioned in the formal characterization of the problem structure. Figure 4 gives an action specification for

```
(:action increase-head
   :parameters (?q - queue ?qt - queuetype
                ?qs1 ?qs2 - queue-state)
   :precondition
     (and
        (next ?qt ?qs1 ?qs2) (is-a-queue ?q ?qt) (head ?q ?qs1)
        (>= (size ?q) 1) (advance-head ?q))
   :effect
     (and
        (settled ?q)  (head ?q ?qs2)
        (not (head ?q ?qs1)) (not (advance-head ?q))
        (assign (head-value ?q) (queue-value ?q ?qs2))
        (assign (head-msg ?q) (queue-msg ?q ?qs2))
        (decrease (size ?q) 1)))
```

Fig. 5. Increasing the head pointer to settle the queue.

reading a variable v from the queue q in transition t querying message m, i.e. $Q?m(v)$. The PDDL representation of $Q!m(v)$ is analogous.

We can see that the state transition enabling flag `ok` is set. The according queue update `increase-head` is shown in Figure 5. As the name indicates it actualizes the head position and eliminates the *settled* flag, which is preconditioned in any queue access action. To disallow actions to be activated twice, before an action is performed we additionally precondition the `active-trans` and `perform-trans` with the settlement of all queues. The required `(forall (q - queue) (settled ?q))` construct can be avoided by removing the queue parameter in the predicate `settled`.

4.4 Variable Handling

As said, variable conditioning and updating is more difficult than other operations, since they require both changes to the instance and problem domain file.

To tame the number of actions, for each condition or assignment the parser generates a pattern structure. For example, setting variable `Active[0]` to 1 corresponds to a `V0=1` pattern. The assignment of any variable to the content of another corresponds to a `V0=V1` pattern.

Conditions on `else` branches are made explicit by the negation of the condition at the corresponding `if` branch.

Inferred patterns generate actions and initial state patterns. E.g. `V0=V1` generates a `is-V0=V1` predicate, to be grounded in the initial state for each variable-to-variable assignment according to the given transition and process for the initial state. The inferred action declaration for the domain file is shown in Table 6.

The `inside` predicate avoids a fourth parameter in the `is-V0=V1` predicate[2].

[2] Actually, this modeling turned out to be crucial for the planner to parse the code

```
(:action V0=V1
   :parameters (?p - process ?t - transition ?v0 ?v1 - variable)
   :precondition
      (and
        (activate ?p ?t) (inside ?p ?t ?v0) (inside ?p ?t ?v1)
        (is-V0=V1 ?t ?v0 ?v1))
   :effect
      (and
        (ok ?p ?t)  (not (activate ?p ?t))
        (assign (value ?v0) (value ?v1))))
```

Fig. 6. Reading variables from a queue.

5 The Plan Relaxation Heuristic

As an example of possible gains of the compilation process we next explain the probably most influencing heuristic in action planning.

The relaxation a^+ of an action $a = (\mathit{pre}(a), \mathit{eff}(a)^+, \mathit{eff}(a)^-)$ is defined as $a^+ = (\mathit{pre}(a), \mathit{eff}(a)^+, \emptyset)$. The relaxation of a planning problem is the one in which all actions are substituted by their relaxed counterparts. The following properties are satisfied [16]:

- P1: Any solution that solves the original plan also solves the relaxed one.
- P2: All preconditions and goals can be achieved if and only if they can in the relaxed task
- P3: The relaxed plan can be solved in polynomial time.

Solving relaxed plans can efficiently be done by building a relaxed problem graph, followed by a greedy plan generation process. Table 1 depicts the implementation of the plan construction and the solution extraction phase.

The first phase constructs the layered plan graph identical to the first phase of *Graphplan* [2]. In Layer i all facts are given that are reachable by applying an operator with satisfied precondition facts in any Layer j with $1 \leq j < i$. In Layer 0 we have all facts present in the initial state. Since we have a finite number of grounded propositions the process eventually reaches a fix-point. The next loop marks the goal facts. In the implementation, the fact layer and goal lists are given as bit-vectors with the latter one constructed on-the-fly.

The second phase is the greedy plan extraction phase. It performs a backward search to match facts to enabling operators. The goal facts build the first unmarked facts. As long as there are unmarked facts in Level i, select an operator that makes this fact true and mark all add effects, and queue all preconditions as unmarked new goals. If there is no unmarked fact left in Level i continue with Level $i - 1$ until the only unmarked facts are the initial ones in Layer 0. The heuristic is constructive, i.e. it not only returns the estimated distance but also a corresponding sequence of actions.

Extending the above estimate to numbers has been achieved as follows [15]. Conditions are of the form $(v \oplus c)$, where $\oplus \in \{\geq, >\}$ and $c \in \mathbb{R}$ and assignments are

procedure Relax(S, *goal*)
$P_0 \leftarrow S$; $t \leftarrow 0$
while (*goal* $\not\subseteq P_t$) **do**
$P_{t+1} \leftarrow P_t \cup \bigcup_{\mathit{pre}(a) \subseteq P_t} \mathit{eff}(a)^+$
if ($P_{t+1} = P_t$) **return** ∞
$t \leftarrow t + 1$
for $i \leftarrow t$ **downto** 1
$G_i \leftarrow \{g \in \mathit{goal} \mid \mathit{level}(g) = i\}$
for $i \leftarrow t$ **downto** 1
for $g \in G_i$
if $\exists a.\ g \in \mathit{eff}(a)^+$ **and** $\mathit{level}(a) = i - 1$
$A \leftarrow A \cup \{a\}$
for $p \in \mathit{pre}(a)$
$G_{\mathit{level}(p)} = G_{\mathit{level}(p)} \cup \{p\}$
return $|A|$

Table 1. Relaxed propositional heuristic.

of the form $(v \otimes c)$, where $\otimes \in \{\texttt{+=}, \texttt{-=}\}$ and $c \in \mathbb{R}^+ \setminus \{0\}$. The *restricted numerical task* is obtained by dropping the delete and the decrease effects and by neglecting numerical preconditions. For restricted expressions, conditions P1-P3 can be satisfied, but through the lack of numerical preconditions the restricted language is too limited for our proposes. Therefore, *strongly monotonic actions* are introduced. The conditions are of the form $(e \oplus e')$, with e, e' being expressions, $\oplus \in \{=, \leq, <, \geq, >\}$, so that the conditions prefer larger variable values, and there exists at least one variable assignment that makes the condition true. The assignments $(v \otimes e')$ with $\otimes \in \{\texttt{:=}, \texttt{+=}, \texttt{-=}\}$ require that the value which the effect adds to the affected variable, increases with the variable (ensuring that repeated application diverges the value), and that the expressions diverge in all numerical values.

For monotone expressions P1-P3 are satisfied, yielding the following set of numerical operations $(e \oplus 0)$, $\oplus \in \{\geq, >\}$ and $(v \otimes e)$, where $\otimes \in \{\texttt{:=}, \texttt{+=}\}$ and e is a linear expression. General linear expressions can be dealt with, since conditions of the form $(e \oplus e')$, with e, e' being linear expressions on the set of numerical variables with $\oplus \in \{=, \leq, <, \geq, >\}$, and assignments of the form $(v \otimes e)$, where $\otimes \in \{\texttt{:=}, \texttt{+=}, \texttt{-=}\}$ can be automatically transformed into the ones above.

For the proper implementation of the heuristic estimate, that is: constructing the relaxed planning graph for the monotone set of linear expressions and extracting the relaxed plan, the reader is referred to [15].

6 Experiments

In the experiments, we include the results of the parser as inputs for recent domain-independent planners. The experiments we performed were run on a Linux PC with

with 800 MHz and 128 MByte main memory. We selected *Metric-FF* and *Mips* as two action planners and *Spin* and *HSF-Spin* as two Promela model checkers. *Metric-FF* is suited to mixed propositional and numerical problems and restricts to linear precondition constraints. *Mips* is the more general system and can handle problems with arbitrary precondition trees, actions with durations, while producing plans that are in fact schedules.

Spin is selected to use a DFS and HSF-Spin is selected to use Weighted A* with weight 2 and the formula-based heuristic [7]. Since Spin does not provide the number of expanded nodes, we took the number of stored nodes instead. Both planners apply heuristic search with variants of the relaxed planning heuristic described above. Similar to HSF-Spin for *Mips* we choose A* with weight 2, while *Metric-FF* applies *Enforced Hill-Climbing*. As in usual Hill-Climbing this exploration algorithm commits a change to a global system state that can never be withdrawn. Different to Hill-Climbing the neighborhood of the current state is traversed with a breadth-first search until a node is found that has a smaller h-value than the current one. Even though the algorithm is incomplete for directed graphs it turns out to be efficient in practice.

The planner *Mips* solves the Leader Election example problem with 107 node expansions in 1.5 seconds [3]. The plan length is 58. Spin finds the bug with only 30 nodes stored, while HSF-Spin solves the same problem with 95 node expansions. Both model checker run about a 1/100 second to come up with the according error trail.

Tables 2 and 3 show that the planners are also capable to efficiently solve deadlock problems in pure message passing systems like the Dining Philosopher and the Optical Telegraph protocol. The comparison for this case is not fair, since exact deadlock state descriptions were used for the planners and not for the model checkers. Through the differences in the interpretation of an executed transition, the trail length also do not match. But that's not the point. At the current level of implementation we do not want to challenge refined model checker implementations, but trademark that efficient action planners can indeed model check communication protocols.

In the philosophers example we see that directed search pays off, since *Spin* searches to large depths until it finds an error. Its DFS exploration order misses shallow errors. On the other hand, *Spin* is fast. In the optical telegraph example it runs fastest (less than 1/100 second), while expanding the largest number of nodes. HSF-Spin takes some more time to expand a node, but performs well in both domains, and its efficiency is better than or equal to the fastest planner.

Mips exploration engine turns out to be the slow. However, the number of expanded nodes in both example protocols is at most 1 from the optimal possible. The suboptimal numbers of expansions in the model checkers are due to graph contraction features, expecially due to state merging. The planner *Metric-FF* has a node expansion routine about as fast as HSF-Spin, which is very good, considering that the planner uses propo-

[3] As a current drawback, *Metric-FF* cannot compete with its refined numerical relaxed planning heuristic, since it cannot properly parse the code provided in the Leader's Election protocol, our running example. The problem is that it detects cyclic assign conditions to be problematic. We are currently in contact to the author of the planner to circumvent this difficulty.

	Metric-FF			MIPS			Spin			HSF-Spin		
p	t	l	e	t	l	e	t	l	s	t	l	e
3	0.02	6	7	0.19	6[2]	7	0.00	18	10	0.01	14	17
4	0.02	8	13	0.27	8[2]	9	0.00	54	45	0.01	18	22
5	0.04	10	21	0.31	10[2]	11	0.00	66	51	0.01	22	27
6	0.05	12	31	0.38	12[2]	13	0.01	302	287	0.01	26	32
7	0.06	14	43	0.47	14[2]	15	0.01	330	309	0.01	30	37
8	0.10	16	57	0.56	16[2]	17	0.02	1.362	1,341	0.02	34	42
9	0.12	18	73	0.69	18[2]	19	0.02	1,466	1,440	0.01	38	47
10	0.15	20	91	0.85	20[2]	21	0.12	9,422	9,396	0.01	42	52
11	0.20	22	111	1.01	22[2]	23	0.15	9,382	9,349	0.01	46	46
12	0.27	24	133	1.23	24[2]	25	1.35	9,998	43,699	0.02	50	62
13	0.32	26	157	1.44	26[2]	27	43.06	9,998	722,014	0.01	54	67
14	0.40	28	183	1.75	28[2]	29	o.m	o.m.	o.m	0.03	58	72
15	0.48	30	211	2.05	30[2]	31	o.m	o.m	o.m	0.03	62	58

Table 2. Results in the deadlock solution for the Dining Philosopher protocol (o.m denotes a run exhausting memory resources and the numbers in brackets [] denote parallel plan lengths).

sitional information, computes an involved estimate and does not have any information on the application domain.

7 Conclusions

We have seen a parser that transforms active processes from Promela into PDDL. The parsing process exploits the automata representation produced by Spin for its visualizer XSpin, and shapes off define-directives. The intermediate result is comprehensible by an end user and can serve as an independent alternative to Promela for modeling communication protocols.

The parser is experimental and cannot interpret full Promela specification. Moreover, experimental results were given for simpler-structured protocols only. We currently work on larger protocols starting with the elevator simulator of A. Biere. The domain does not use any queue communication and parses fine except of indirectly referenced variables.

Through the inferred intermediate planner input format, we provide the basis for a host of new algorithmic aspects to the exploration process, bridging the gap between the two disciplines model checking and action planning. For example, new, apparently more informed heuristics can be applied to the validation of protocols.

The paper is ambitious in the sense that it proposes to bypass the model checker with a domain-independent action planner. A fairer comparison would be to devise domain-dependent pruning rules very similar to LTL specifications, concurently checked during the planning process. Examples of these kinds of planners rules are TL-Plan [1] and Tal-Plan [20].

	Metric-FF			MIPS			Spin			HSF-Spin		
n	t	l	e	t	l	e	t	l	e	t	l	e
3	0.14	18	57	0.55	18[5]	12	0.00	23	11	0.09	21	23
4	0.20	24	80	0.73	24[5]	24	0.00	30	14	0.10	28	30
5	0.31	30	122	1.30	30[5]	30	0.00	37	17	0.20	35	37
6	0.41	36	173	1.46	36[5]	36	0.00	44	20	0.36	42	44
7	0.65	42	233	1.80	42[5]	42	0.00	51	23	0.45	49	51
8	0.90	48	302	2.12	48[5]	48	0.00	58	26	0.61	56	58
9	1.12	54	271	2.46	54[5]	54	0.00	65	29	1.07	63	65
10	1.60	60	331	2.85	60[5]	60	0.00	72	32	1.24	70	72
11	2.20	66	397	3.31	66[5]	66	0.00	79	35	1.86	77	79
12	2.85	72	469	3.74	72[5]	72	0.00	86	38	2.32	84	84
13	3.95	78	547	4.22	78[5]	78	0.00	93	41	4.01	91	93
14	5.05	84	631	4.92	84[5]	84	0.00	100	44	4.83	98	100

Table 3. Results for the Optical Telegraph protocol.

The works initiates experiments with concurrency problems in AI planners. The current research focus is the exploitation of similarities and differences of model checkers and planners on comparable benchmark inputs to improve both areas with algorithmic contributions of the other. Our choice to implement the parser was influenced by the fact, that it could be slightly harder to implement the heuristics into an existing model checker than to export the description to PDDL. In the long term we expect some of these estimates to become an integral part of directed model checkers like HSF-Spin [7].

The plan relaxation heurisitic is not the only effective guidiance. We highlight the pattern database heuristic [5], which is to be computed as a large lookup-table prior to the search. This heuristic can be included in BDD exploration engines as shown in the planner *Mips*. Since the estimates were adapted from efficient domain-independent action planners, the design patterns are more general and likely to transfer to the validation of security protocols or other model checking exploration tasks.

In future research we plan to look at probabilistic model checking and planning also, taking factored Markov Decision Processes (FMDPs) as a welcome theoretical fundament. By means of the contributed work, we think of using a FMDP solver like SPUDD [14] and to transfer annotated Promela into an intermediate probabilistic action planning description language. SPUDD takes algrebraic decision diagrams (ADDs) for internal state representation structure and performs the Bellman update for real-time dynamic programming symbolically. We will also look at symbolic heuristic search for FMDP by joining And/Or-tree search with the SPUDD approach.

*A*cknowledgements The author would like to thank DFG for support in the projects *Heuristic Search and its Application to Protocol Verification*, *Directed Model Checking (with AI exploration algorithms)* and *Heuristic Search*. Thanks to Alberto Lluch-Lafuente for providing appropriate Promela benchmark models. Thanks to Tilman Mehler for proofreading.

References

1. F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116:123–191, 2000.
2. A. Blum and M. L. Furst. Fast planning through planning graph analysis. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 1636–1642, 1995.
3. B. Bonet and H. Geffner. Planning as heuristic search. *Artificial Intelligence*, 2001.
4. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
5. S. Edelkamp. Planning with pattern databases. In *European Conference on Planning (ECP)*, 2001. 13-24.
6. S. Edelkamp. Taming numbers and durations in the model checking integrated planning system. *Journal of Artificial Research (JAIR)*, 2003. Submitted, A draft is available at PUK-Workshop 2002.
7. S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology (STTT)*, 2002.
8. R. Fikes and N. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.
9. M. Fox and D. Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. Technical report, University of Durham, UK, 2001.
10. A. Gerevini and L. Schubert. Discovering state constraints in DISCOPLAN: Some new results. In *National Conference on Artificial Intelligence (AAAI)*, pages 761–767, 2000.
11. A. Gerevini and I. Serina. LPG: a planner based on local search for planning graphs with action costs. In *Artificial Intelligence Planning and Scheduling (AIPS)*, pages 13–22, 2002.
12. P. Godefroid and S. Khurshid. Exploring very large state spaces using genetic algorithms. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2002.
13. P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for heuristic determination of minimum path cost. *IEEE Transactions on on Systems Science and Cybernetics*, 4:100–107, 1968.
14. J. Hoey, R. Aubin, A. Hu, and C. Boutilier. Spudd: Stochastic planning using decision diagrams. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, 1999.
15. J. Hoffmann. Extending FF to numerical state variables. In *European Conference on Artificial Intelligence*, 2002.
16. J. Hoffmann and B. Nebel. Fast plan generation through heuristic search. *Artificial Intelligence Research*, 14:253–302, 2001.
17. G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1990.
18. G. J. Holzmann and M. H. Smith. Software model checking: Extracting verification models from source code. In *Formal Description Techniques for Distributed Systems and Communication Protocols, Protocol Specification, Testing and Verification (FORTE/PSTV)*, pages 481–497, 1999.
19. R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.
20. J. Kvarnström, P. Doherty, and P. Haslum. Extending TALplanner with concurrency and ressources. In *European Conference on Artificial Intelligence (ECAI)*, pages 501–505, 2000.
21. D. Long and M. Fox. Efficient implementation of the plan graph in STAN. *Artificial Intelligence Research*, 10:87–115, 1998.
22. J. Pearl. *Heuristics*. Addison-Wesley, 1985.
23. C. H. Yang and D. L. Dill. Validation with guided search of the state space. In *Conference on Design Automation (DAC)*, pages 599–604, 1998.